Memory and compiler optimizations for low-power and -energy.

Olivier Zendra

Memory and compiler optimizations for low-power and -energy

Olivier Zendra

INRIA-Lorraine / LORIA, Building C,
615 Rue du Jardin Botanique, BP 101,
54602 Villers-Lès-Nancy Cedex,
FRANCE
Olivier.Zendra@loria.fr
http://www.loria.fr/~zendra

Abstract. Embedded systems become more and more widespread, especially autonomous ones, and clearly tend to be ubiquitous. In such systems, low-power and low-energy usage get ever more crucial. Furthermore, these issues also become paramount in (massively) multi-processors systems, either in one machine or more widely in a grid. The various problems faced pertain to autonomy, power supply possibilities, thermal dissipation, or even sheer energy cost.
Although it has since long been studied in harware, energy optimization is more recent in software. In this paper, we thus aim at raising awareness to low-power and low-energy issues in the language and compilation community. We thus broadly but briefly survey techniques and solutions to this energy issue, focusing on a few specific aspects in the context of compiler optimizations and memory management.

1 Introduction

Embedded systems become more and more widespread, especially autonomous ones, and clearly tend to be ubiquitous. In such systems, low-power and low-energy usage get ever more crucial. Furthermore, these issues also become paramount in (massively) multi-processors systems, either in one machine or more widely in a grid. The various problems faced pertain to autonomy, power supply possibilities, thermal dissipation, or even energy cost.

They can be addressed from various viewpoints, either in hardware or software. Among those, hardware design implies work at the microelectronics and physics level, while hardware optimization incurs on-line logic, dedicated circuits and thus some overhead at runtime (energy and/or time).

Software optimization at compile-time, on the other side, is an off-line logic (if performed statically) that incurs no runtime overhead. It can also use from more resources (time, memory...), which allows larger contexts. It may however have trouble capturing the exact runtime behavior of a system.

Energy considerations are relatively recent in compilation, which historically focused more on size and speed. Optimizations for speed and energy are often

related [12] but not always. For example, moving some work out of the critical path is good for time, but it may simply not impact energy, since the work still has to be done. It is also worth stressing that optimizing for (peak) power is different from optimizing for energy (average power over time) and different from optimizing for power density (temperature and hot spots).

In this paper, we aim at raising awareness to low-power and low-energy issues in the language and compilation community. We thus perform a broad but shallow survey of techniques and solutions, focusing on a few specific aspects, all in the context of compiler optimizations and memory management.

2 Transitions and commutations

One of the lowest levels at which compilation can address energy issues is the bit level. Transitions (bit commutations) between successive instructions cost energy. A compiler can reschedule instructions to minimize this cost [7, p.193].

One example is register renaming in order to decrease commutations for the register name field of instructions. Commutation activity on this field was reduced by 11% in [11]. The overall, global impact of this specific optimization has nonetheless to be evaluated at whole program level.

3 Loop optimizations

Very numerous research works have revolved around loop optimizations. These were historically targeted to speed. Detailing all these works falls beyond the scope and space of this paper of this paper, but we can mention a few ones.

One canonical example is loop unrolling, where 1 loop with length n running i times becomes 1 loop with length $n * x$ running n/x times:

```
for(i=0;i<10000;i++){                          for(i=0;i<5000;i++){
   a();b();            ==== loop unrolling ===>    a();b();
}                                                  a();b();  }
```

The impact of loop unrolling is twofold. First, the static instruction count increases, since some instructions are duplicated. This leads to a larger code size, hence an increase in energy usage. But a second effect is to decrease the number of dynamic instructions, since less are executed for loop control. This translates into a gain in time as well as in energy. It is thus important to balance overhead and gain when using loop unrolling in an energy-sensitive context.

More examples of loop-oriented optimization for energy will follow in the context of memory management.

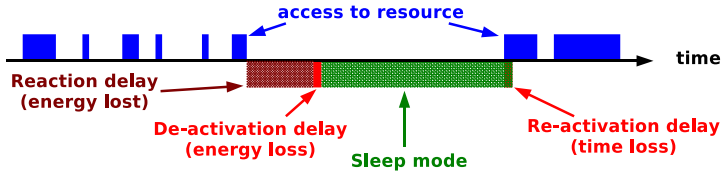
4 Execution modes

This idea behind program modes is to follow program phases. The most famous example of execution mode is DVS/DFS (Dynamic Voltage Scaling / Dynamic

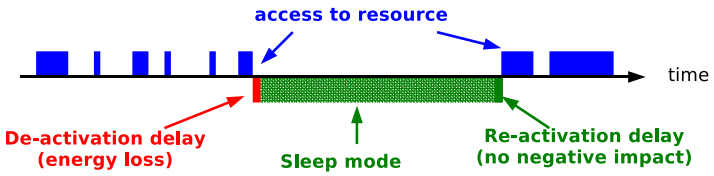
Frequency Scaling), which deals with the CPU. Since $P = C.V^2.f$ and $E = P_{avg}.Time$, DVFS consists in decreasing voltage and frequency (both are tied) to save (dynamic) power and energy.

Other resources also feature execution modes targeted to low energy, such as sleep modes, or hibernation. Using these modes aims at 0 consumption (unused resource). Both dynamic and static power are concerned, which is very effective.

Hardware is able to easily detect phases of low utilization of one resource, but only *a posteriori*. It has more difficulties and less certainty when trying to *predict* future low usage phases. This incurs useless delay before appropriate action (put into some sleep mode) is taken, as the following figure shows:.



A compiler, on the contrary, can spot the points where a resource is (going to be) unused. The latter thus can be put into sleep mode immediately. Furthermore, a compiler can even "warn" in advance the hardware of a future sleep period for a resource, as well as of a future re-start. There is thus no unneeded delay when going to sleep mode or waking-up a resource, which provides much better results, both in terms of energy and time:



In the following sections, we will develop on the role of compilation and memory management to take advantage of sleep modes.

5 Register window

Register windows consist in having more virtual register than actual (physical) ones, the virtual registers being separated in several sets called "register windows". Of course, only 1 register window can be active (used) at a time.

The underlying principle is to change register window according to program phases: one phase runs into one window. This makes it possible to reduce register spill (use of main memory when not enough registers are available), at the cost a some management overhead (to handle window changes, registers are swapped with memory).

Working with registers rather than memory means decreasing the number of transfers and increasing speed (the very reason register windows were created for), by up to 11% according to [14].

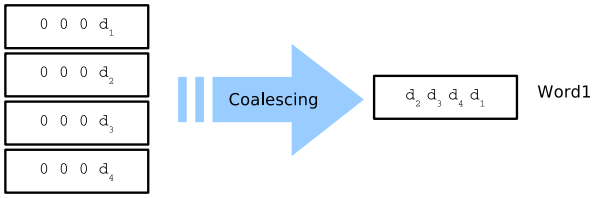
But this also offers some increased opportunities for sleep modes. In terms of energy, register windows can thus lead to important saving, up to 25% [14].

6 Memory management: compaction

The idea behind compaction for low energy is very basic: less space generally means less energy (to hold code or data), and possibly more opportunities for memory banks sleep modes. There are many ways to have compact information.

The first one is **contiguity**, which consists in allocating and keeping data in a minimum of well-filled areas, the others being put into low-power mode. To achieve this goal, data moves may be necessary to avoid fragmentation. Note that contiguity may be antagonist to speed, since the latter may benefit from parallel accesses to several memory banks.

Coalescing is a way to decrease the space taken by pieces of data. Coalescing variables consists in fitting several "small" pieces of data into only 1 slot. Subword data and bitwidth aware register allocation are examples of "spatial" data coalescing.



Lifetime analysis is a dual approach allowing "temporal coalescing". It consists in putting in the same slot pieces of data that do not coexist at the same time, even though each piece completely fills the slot.

Code or data **compression** is well known and works on a much larger scale. It thus offers stronger opportunities to decrease size (especially for data), hence helping increase sleep modes usage. However, the potential overhead of compression is high. Compression thus appears more adapted to long-lived and seldom accessed data.

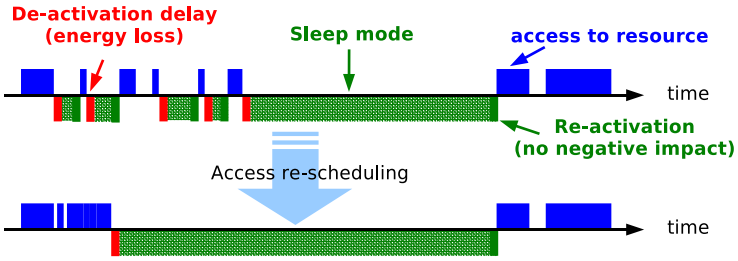
Overall the impact of compaction techniques can be very significant. On variables,[19] reports 3% less cycles and 69% smaller stack, while [15] reaches 10 to 50% saving in the number of registers needed. In [18], data (fields) compression allows 25% decrease in heap size and 30% in energy, while runtime decreases by 12% and even 30% when ISA Data Compression eXtensions are available.

7 Resource access scheduling

Another very effective way of saving energy in software is to improve the locality of accesses to resources. Indeed, grouping accesses increases the length of the

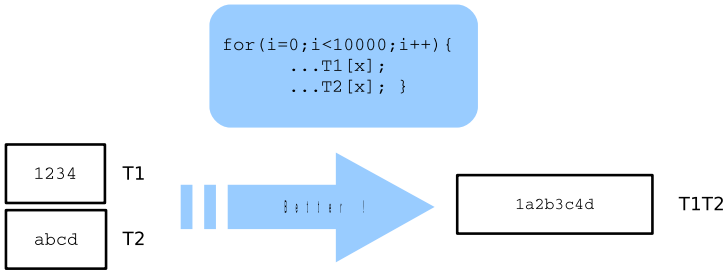
periods over which a specific resource is unused, and can thus be put into some low-power sleep mode.

Access re-scheduling occurs at code level, and consist in clustering and advancing some accesses:



A specific case of access re-scheduling can be performed at loop level. Loop fission consists in splitting 1 loop into several loops, when the instructions in the original loop do not depend on each other. The various loop can thus process different pieces of data (arrays...), which offers a better locality and increases sleep mode opportunities [7, ch10].

Access re-scheduling may also be performed at the data level, by changing data layout. This is dual of code change and is especially interesting for arrays. Indeed, accessing those according to layout makes it possible not only to save time but also to reach up to 10% saving in energy wrt. to basic mode control [2]. Similarly, interlacing arrays that are accessed simultaneously, as shown below, offers 8% savings in energy over basic mode control [2]:



8 Scratch-pad memory

Memory-related issues can be of tremendous importance when trying to save energy. Some studies even claim that 70% to 90% of energy in 2010 will be used by memory [10].

It is well known that caches generally provide great speed gains. However, they are poorly adapted to embedded systems, especially autonomous ones. They indeed increase the circuit size (for the cache memory as well as its management

logic), are very energy-hungry and are poorly predictable, which is an issue with real time systems. Many cacheless systems thus remain in the embedded world.

Scratch-Pad Memory (SPM) is intended to avoid the main drawbacks of caches. They consists of small, fast memory areas (SRAM...), very much like caches, but are directly and explicitly managed at the software level, either by the developer or by the compiler. Hence, no dedicated circuit is required for SPM management.

Their advantages compared to caches are numerous [4]. SPM takes up to 34% less area than a cache: only the memory is present, without additional logic. Their cost is thus lower. Furthermore, their explicit management makes them more predictable for real-time systems. Finally, SPM uses up to 40% less energy than caches.

SPM can be used throughout a wide range of application domains. They perform greatly if data accesses are known and regular, which is for example the case in matrix multiply, audio-video compression algorithms, filtering... They also perform well — and better than caches — if the mapping of data in the SPM is optimal based on access probabilities. This has been proven in [1] for structures like lists, n-trees with low-variation topology

The management of the SPM significantly impacts how it performs.

Static management is a first kind of SPM management strategy. There, choices (that is data placements) are performed entirely off line, at compile time, and no data move ever occurs. These strategies can nonetheless take into account some runtime information thanks to execution profiles. They generally feature good performance and good real time characteristics.

Dynamic strategies are more complex and more recent but have a tremendous potential. There, the allocation into the SPM is dynamic, thats is performed at runtime, although it is generally decided at compile time. (Dis)placements may occur at runtime. The main interest of dynamic SPM management techniques lies in the fact they can take into account regions in the program, or program phases, instead of considering the program as a whole. Allocation choices are based on usage frequency, transfer costs (between SPM and memory), size of objects...

A dynamic management of SPM has many advantages. It allows better memory (re)use, since freeing SPM immediately is possible when a piece of data ceases (maybe temporarily) to be in use. It is also better on more complex situations (MPEG21, MPEG4), with dynamic creation of tasks, variable data size... However, the dynamic management of SPM tends to make real time constraints harder to meet. Furthermore, it increases a bit the size of the program, because of the (software) logic needed. Dynamic SPM management also incurs an overhead, both in terms of energy and time, compared to static SPM management. The dynamic logic is indeed more complex, and transfers between SPM and RAM are expensive. Note that this cost can to some extend be decreased with DMA support [13], or eliminated completely when direct allocation in SPM (without coming first from RAM) is possible.

Overall, dynamic management of SPM results in 35% runtime and 40% energy savings wrt. a static placement (except heap) in SPM.

9 Global system analysis

It is well-known in compilation that system-wide analysis makes it possible to perform more aggressive and precise (inter-program) optimizations. In the specific context of energy, this holds true as well.

As an example, [9] considered buffer sizing and access clustering. They were able to reduce energy usage by 7% to 49% with multi-program optimization wrt. mono-program optimization.

10 Conclusion and perspectives

Hardware and compilation complete each other. Compilation can take into account a much larger context, since it has lots of resources (especially static compilation). But it may not catch the very exact runtime behavior of programs, unlike hardware. Trying to have the best of both worlds is tenting: optimizing Virtual Machines do it. But they remain expensive in terms of resources at runtime.

We think there is thus a clear need of support for hardware-software (compiler) interface at the ISA level, in order to increase the synergies between the two worlds. This way, "direct" management of resources by the compiler would be possible, as well as co-optimizations involving compiler and hardware, with information transmissions both ways.

VLIW processors and the EPIC architecture offer a higher potential of parallelism. This would of course increase speed, be can also be of interest energy-wise. The problem with these architectures is that the compiler has to provide the parallelism, with requires a lot of work not yet done for generic processors.

Finally, we mentioned the paramount importance of memory for energy usage optimization. We want once again to stress the immediate potential of SPM (scratch-pad memory), as well as the — possibly longer-term — potential of Energy-aware Garbage Collectors.

References

1. Mohammed Javed Absar and Francky Catthoor. Compiler-based approach for exploiting scratch-pad in presence of irregular array access. In *DATE*, pages 1162–1167, 2005.
2. R. Athavale, Narayanan Vijaykrishnan, Mahmut T. Kandemir, and Mary Jane Irwin. Influence of array allocation mechanisms on memory system energy. In *IPDPS*, page 3, 2001.
3. Oren Avissar, Rajeev Barua, and Dave Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *Transaction. on Embedded Computing Systems.*, 1(1):6–26, 2002.

4. Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *10th international symposium on Hardware/software codesign (CODES'02)*, pages 73–78, New York, NY, USA, 2002. ACM Press.
5. V. Delaluz, M. Kandemir, N. Vijaykrishnan, M. J. Irwin, A. Sivasubramaniam, and I. Kolcu. Compiler-directed array interleaving for reducing energy in multi-bank memories. In *2002 conference on Asia South Pacific design automation/VLSI Design (ASP-DAC'02)*, page 288, Washington, DC, USA, 2002. IEEE Computer Society.
6. Angel Dominguez, Sumesh Udayakumaran, and Rajeev Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Computing (JEC)*, 1(4), 2005.
7. Robert Graybill and Rami Melhem. *Power aware computing*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.
8. J. Hom and U. Kremer. Energy management of virtual memory on diskless devices. In *Workshop on Compilers and Operating Systems for Low Power (COLP'01)*, Barcelone, Espagne, September 2001.
9. Jerry Hom and Ulrich Kremer. Inter-program optimizations for conserving disk energy. In *2005 international symposium on Low power electronics and design (ISLPED'05)*, pages 335–338, New York, NY, USA, 2005. ACM Press.
10. ITRS. International technology roadmap for semiconductors, 2005. http://public.itrs.net.
11. M. Kandemir, N. Vijaykrishnan, M.J. Irwin, W. Ye, and I. Demirkiran. Register relabeling: A post compilation technique for energy reduction. In *Workshop on Compilers and Operating Systems for Low Power (COLP'00)*, Philadelphie, PA, USA, October 2000.
12. M. Lee, V. Tiwari, S. Malik, and M. Fujita. Power analysis and minimization techniques for embedded dsp software. *IEEE Transactions on Very Large Scale Integration*, 5, March 1997.
13. Francesco Poletti, Paul Marchal, David Atienza, Luca Benini, Francky Catthoor, and Jose Manuel Mendias. An integrated hardware/software approach for run-time scratchpad management. In *DAC*, pages 238–243, 2004.
14. Rajiv A. Ravindran, Robert M. Senger, Eric D. Marsman, Ganesh S. Dasika, Matthew R. Guthaus, Scott A. Mahlke, and Richard B. Brown. Partitioning variables across register windows to reduce spill code in a low-power processor. *IEEE Transaction on Computers*, 54(8):998–1012, 2005.
15. Sriraman Tallam and Rajiv Gupta. Bitwidth aware global register allocation. In *POPL*, pages 85–96, 2003.
16. Seungdo Woo, Jungroin Yoon, and Jihong Kim. Low-power instruction encoding techniques. In *SOC Design Conference*, 2001.
17. Fen Xie, Margaret Martonosi, and Sharad Malik. Intraprogram dynamic voltage scaling: Bounding opportunities with analytic modeling. *ACM Transactions on Architure and Code Optimization (TACO)*, 1(3):323–367, 2004.
18. Youtao Zhang and Rajiv Gupta. Data compression transformations for dynamically allocated data structures. In *11th International Conference on Compiler Construction (CC'02), Lecture Notes in Computer Science*, volume 2304, pages 14–28, London, UK, 2002. Springer-Verlag.
19. Xiaotong Zhuang, ChokSheak Lau, and Santosh Pande. Storage assignment optimizations through variable coalescence for embedded processors. In *LCTES '03: 2003 ACM SIGPLAN conference on Language, Compiler, and Tool for Embedded Systems*, pages 220–231, New York, NY, USA, 2003. ACM Press.